Lektion 6

In dieser Lektion geht es nun um die Frage, wie die Effizienz von Algorithmen definiert und analysiert werden kann. Sie lernen das grundlegende Verfahren dafür kennen, wie die Komplexität von Algorithmen angegeben wird. Die sog. Landau-Notation, die dazu verwendet wird, gibt nur das prinzipielle Größenwachstum von Funktionen an, d.h. das Wachstumsverhalten für sehr große Werte bei Problemgröße n gegen Unendlich. Auf diese Weise ist es möglich, Eigenschaften des Laufzeitverhaltens auf eine Weise zu beschreiben, die unabhängig von der konkreten Ausführungsgeschwindigkeit eines Rechners ist.

4.4 Komplexität von Algorithmen

Wenn geklärt ist, dass ein Algorithmus korrekt ist, kann im nächsten Schritt betrachtet werden, wie effizient der Algorithmus ist. Man spricht in diesem Zusammenhang von der *Komplexität* eines Algorithmus.

Definition 4.19 - Zeit- und Platzkomplexität

Komplexität bezeichnet den Ressourcenverbrauch von Algorithmen. Ressourcen sind dabei:

(1) Ausführungszeit (**Zeitkomplexität**)

(2) Speicherbedarf (**Platzkomplexität**)

Einflussfaktoren für Komplexität

Der Ressourcenbedarf hängt von mehreren Faktoren ab:

- Umfang der Daten (Größe des Problems): z.B. sind nur wenig oder viele Werte zu sortieren.
- Zusammensetzung der Daten: z.B. sind die zu sortierenden Daten schon teilweise in richtiger Ordnung oder sind sie völlig "durcheinander"?
- Ausführungsgeschwindigkeit und Speicherbedarf bei der Ausführung auf einem Rechner: hängt natürlich vom verwendeten Rechnermodell, der Programmiersprache und der Implementierung der Programmiersprache ab.

4.4.1 Untersuchung des Ressourcenverbrauchs

Will man eine Aussage über die Komplexität eines Algorithmus machen, sollte das unabhängig von einem konkreten Rechner und einer konkreten Programmiersprachen sein. Z.B. eine Aussage wie

> "*Insertionsort für 1 Mio. zufällig gewählte Werte braucht auf einem Rechner-Modell XXX mit y GB Hauptspeicher in der Java-Implementierung t Millisekunden Laufzeit*"

ist wenig hilfreich, wenn allgemeine Eigenschaften des Insertionsort-Algorithmus beschreiben werden sollen.

Für Komplexitätsuntersuchungen geht man deshalb für die Analyse von einem abstrakten Maschinenmodell aus.

RAM-Maschinenmodell für die Komplexitätsanalyse

Als Maschinenmodell wird die **Random-Access Machine (RAM)** - Maschine mit wahlfreiem Speicherzugriff verwendet:

- Eine RAM ist ein idealisierter Von-Neumann-Rechner
- Random-access heißt: frei adressierbare Speicherzellen, d.h. jede Speicherzelle kann zu jeder Zeit mit gleichem Aufwand angesprochen werden, so wie der Hauptspeicher in einem Rechner (Speicherzugriffe sind also nicht auf sequentielle Zugriffe wie bei einem Speicherband beschränkt).
- Sequentielle Ausführung (1 Prozessor). Parallele Ausführung auf mehrern Prozessoren oder Prozessorkernen wird in Rahmen dieser Vorlesung nicht betrachtet.
- Einmalige Ausführung einer Anweisung A_i erfordert immer die gleiche konstante Zeit c_i (spezifische Eigenschaft der Maschine)

In erster Näherung kann dieses RAM-Modell als eine passende Abstraktion für übliche Rechner betrachtet werden: Ein Programm besteht aus einzelnen Befehlen, die sequentiell nacheinander ausgeführt werden. Jeder Befehl hat eine feste Ausführungszeit. Jede Speicherzelle ist frei adressierbar.

Aufgabe 4.20 - Vergleich RAM und reale Rechner

Überlegen Sie sich, inwieweit das RAM-Modell zur Architektur realer, aktueller Rechner passt. Welche Übereinstimmungen, welche Unterschiede gibt es.

Vorgehen bei der Laufzeitanalyse

Haben wir nun einen Algorithmus und wollen die Laufzeit analysieren, können wir in folgender Weise vorgehen (natürlich nur gedanklich, da die RAM keine reale Maschine ist). Wir nehmen an, wir haben eine bestimmte Problemgröße *n* gegeben.

- Bestimme für jede Anweisung A_i des Programms die Häufigkeit k_i, wie oft sie ausgeführt wird
- Die **Gesamtkosten** (Gesamtlaufzeit) bei Problemgröße *n* können dann einfach entsprechend folgender Summe zusammengezählt werden.

$$T(n) = \sum_{\text{Anweisungen } A_i} c_i \cdot k_i$$

wobei

k_i	Häufigkeit für die Ausführung von Anweisung A_i
c_i	Einzel-Ausführungszeit für Anweisung A_i

d.h. man betrachtet für jede Anweisung, wie oft sie ausgeführt wird, berechnet durch Multiplikation mit der Ausführungszeit c_i die Gesamtzeit, die für diese einzelne Anweisung insgesamt aufgewendet wird, und summiert das über alle Anweisungen auf.

Wir führen das nun einmal so am Beispiel Insertionsort durch. (Schon vorab: Später werden wir es nie mehr auf dieser detaillierten Ebene machen).

Beispiel 4.21 - Analyse von Insertionsort

```
1                 static void insertionSort(double[] arr) {
2   c2            int j = 1;
3   c3            while (j < arr.length) {
4
5
6   c6                double value = arr[j];
7
8   c8                 int insertPos = j;
9   c9                 while(insertPos > 0 && arr[insertPos-1] > value) {
10  c10                          arr[insertPos] = arr[insertPos-1];
11  c11                          insertPos--;
12                           }
13  c13                      arr[insertPos] = value;
14  c14               j++;
15                    }
16                }
```

Kosten der Anweisung

Wir betrachten jeweils eine Zeile mit ausführbaren Java-Konstrukten als eine Anweisung. Da wir die Ausführungszeit (die Kosten) dafür nicht wissen, führen wir dafür jeweils eine Konstante ein, d.h. c_i ist die Zeit, um einmal den Code in Zeile *i* auszuführen.

Die Problemgröße *n* ist hier die Länge des Felds `arr`, das sortiert werden soll.

Kosten und Häufigkeit der Anweisungen bei Feldgröße *n*: Wir können nun eine Tabelle mit den Ausführungshäufigkeiten für die einzelnen Anweisungen aufstellen. Beispielsweise wird Zeile 2 einmal ausgeführt und Zeile 3 insgesamt *n* mal (nicht vergessen, dass nach dem letzten Schleifendurchlauf die Schleifenbedingung nochmal geprüft wird).

Die äußere Schleife ist einfach zu behandeln: Sie wird immer *n*-1 mal durchlaufen, unabhängig vom Inhalt des Arrays. Bei der inneren Schleife haben wir aber ein Problem. Wie oft die innere Schleife zum Einfügen des nächsten Werts durchlaufen wird, hängt von den Daten im Feld ab - und diese Daten kennen wir nicht!

Wenn wir die Zahl der Durchläufe nicht wissen und trotzdem weitermachen wollen, führen wir einfach Konstanten t_j dafür ein.

- t_j ist die Anzahl der Durchläufe der inneren Schleife bei Wert j der äußeren Schleife. Die Schleifenbedingung wird dann (t_j+1) mal geprüft.

Damit können wir nun die Tabelle komplett aufstellen:

Zeile	Kosten	Häufigkeit
2	c_2	1
3	c_3	n
6	c_6	n-1
8	c_8	n-1
9	c_9	$\sum_{j=1}^{n-1}(t_j+1)$
10	c_{10}	$\sum_{j=1}^{n-1} t_j$
11	c_{11}	$\sum_{j=1}^{n-1} t_j$
13	c_{13}	n-1
14	c_{14}	n-

Die Summe für die **Gesamtkosten** zu berechnen, ist jetzt nur noch mühsame Fleißarbeit:

$$T(n) = c_2 + c_3 n + (c_6 + c_8 + c_{13} + c_{14})(n-1) + c_9 \cdot \sum_{j=1}^{n-1} (t_j + 1) + (c_{10} + c_{11}) \cdot \sum_{j=1}^{n-1} t_j$$
$$= (c_2 - c_6 - c_8 - c_{13} - c_{14}) + (c_3 + c_6 + c_8 + c_{13} + c_{14}) n + c_9 \cdot \sum_{j=1}^{n-1} t_j + c_9 \cdot \sum_{j=1}^{n-1} 1 + (c_{10} + c_{11}) \cdot \sum_{j=1}^{n-1} t_j$$
$$= (c_2 - c_6 - c_8 - c_9 - c_{13} - c_{14}) + (c_3 + c_6 + c_8 + c_9 + c_{13} + c_{14}) n + (c_9 + c_{10} + c_{11}) \cdot \sum_{j=1}^{n-1} t_j$$
$$= a + b \cdot n + c \cdot \sum_{j=1}^{n-1} t_j$$

für entsprechende Konstanten a, b, c

$$a = c_2 - c_6 - c_8 - c_9 - c_{13} - c_{14}$$
$$b = c_3 + c_6 + c_8 + c_9 + c_{13} + c_{14}$$
$$c = c_9 + c_{10} + c_{11}$$

Nützt das mühsam erarbeitet Ergebnis uns irgendetwas, um zu verstehen, wie effizient Insertionsort ist? Wir kennen weder die Konstanten c_i für die Ausführungszeiten der Anweisungen noch die Werte t_j für die von den Daten abhängige Anzahl der Wiederholungen der inneren Schleife!

Abhängigkeit von der Datenzusammensetzung

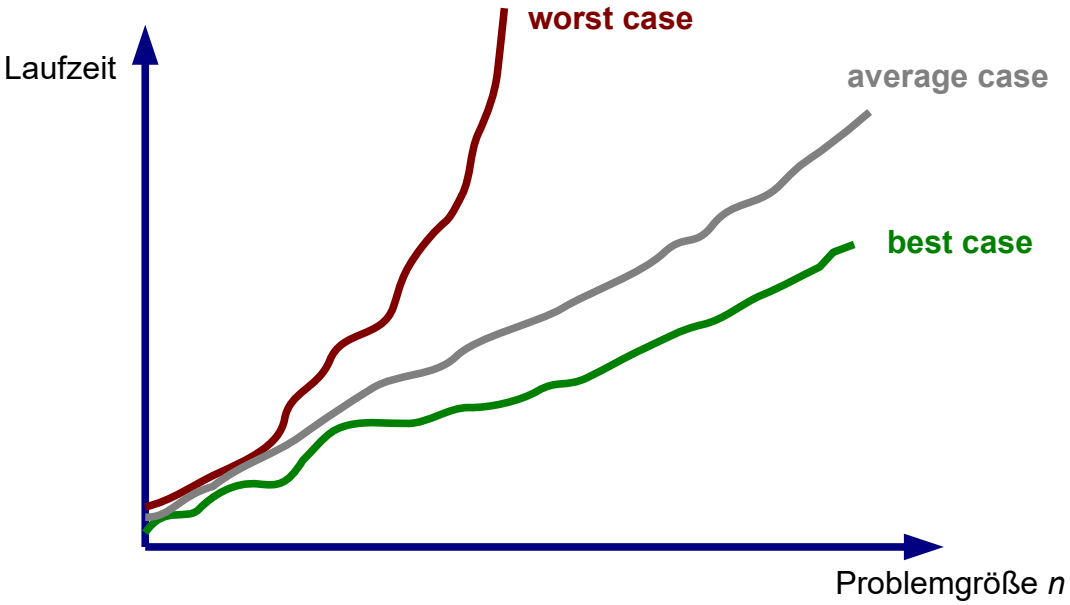
Die Konstanten t_i hängen von der Zusammensetzung der Daten im Array ab. Hat man *n* verschiedene Werte, dann gäbe es *n*! viele unterschiedliche Möglichkeiten dafür. Für große *n* sind das gigantische Zahlen. Da man nicht alles einzeln untersuchen kann, beschränkt man sich üblicherweise bei der Analyse auf drei Fälle:

Bester Fall (best case):
- basiert auf Daten, die im Algorithmus eine **minimale Anzahl von Schritte** erfordern
  → größte untere Schranke für die Laufzeit

Schlechtester Fall (worst case):
- basiert auf Daten, die im Algorithmus eine **maximale Anzahl von Schritten** erfordern
  → kleinste obere Schranke für die Laufzeit

Mittlerer Fall (average case):
- Es wird die **mittlere Anzahl der Schritte** gezählt, gemittelt über alle möglichen Eingabedaten
  → Mittelwert der erwarteten Laufzeiten

Laufzeit in Abhängigkeit von der Problemgröße



Anmerkung

- Üblicherweise wird bei der Average-Case-Analyse eine *Gleichverteilung der Daten* angenommen (sofern nicht anders angegeben), d.h. jede mögliche Datenverteilung ist gleich wahrscheinlich.
- Bei realen Anwendungsszenarien ist dies nicht immer so, so dass dann genauer untersucht werden muss, wie sich die ungleiche Datenverteilung auswirkt.
- Da für die Average-Case-Analyse oft nicht alle möglichen Datenverteilungen analysiert werden können, um dann den Durchschnitt zu berechnen, wird oft so vorgegangen, dass "mittlere" Daten betrachtet werden, d.h. die Situation für zufällig gewählte Daten untersucht wird. Bei einer angenommenen Gleichverteilung führt das dann zu entsprechenden Ergebnissen.

Beispiel 4.22 - Laufzeitanalyse Insertionsort

Ausgehend von der zuvor erfolgten Analyse können wir nun die drei Grundfälle für Insertionsort untersuchen:

- **Günstigster Fall (best case)**: Die Daten sind schon aufsteigend sortiert. Beim Einfügen sind alle Werte bereis an der richtigen Stelle, so dass die innere Schleife nicht durchlaufen werden muss, es gilt also $t_j = 0$ für alle j . Die Summenformel vereinfacht sich dann folgendermaßen:

$$T_{best}(n) = a + b \cdot n + c \cdot \sum_{j=1}^{n-1} 0 = a + b \cdot n$$

Wir kennen die Konstanten *a* und *b* zwar weiterhin nicht, aber wir erkennen zumindest, dass die Laufzeit in diesem Fall eine **lineare Funktion** ist.

- **Ungünstigster Fall (worst case)**: Der ungünstigste Fall ergibt sich für Insertionsort, wenn die Daten absteigend (d.h. umgekehrt) sortiert sind. Jeder einzufügende Wert muss dann immer bis an den Anfang des Arrays vorgeschoben werden. D.h. für die Anzahl der Wiederholungen der inneren Schleife zum Einfügen von arr[j] gilt: $t_j = j$.

  Für die Gesamtlaufzeit ergibt sich dann folgende Formel:

$$T_{worst}(n) = a + b \cdot n + c \cdot \sum_{j=1}^{n-1} j = a + b \cdot n + c \cdot \frac{n \cdot (n-1)}{2} = a + (b - \frac{c}{2}) \cdot n + \frac{c}{2} n^2$$

  Wir haben dabei die Gaußsche Summenformel zur Berechnung der Summe von 1 bis *n*-1 verwendet.

  Auch hier kennen wir zwar die Konstanten *a*, *b* und *c* nicht. Wir sehen aber, dass es eine **quadratische Funktion** ist.

- **Mittlerer Fall (average case)**: Wir gehen davon aus, dass völlig zufällig gemischten Daten beim Einfügen des Werts arr[j] im Mittel bis zur Hälfte des sortierten Bereichs durchgegangen werden muss, d.h. $t_j = j/2$.

$$T_{avg}(n) = a + b \cdot n + c \cdot \sum_{j=1}^{n-1} \frac{j}{2} = a + b \cdot n + \frac{c}{2} \cdot \frac{n \cdot (n-1)}{2} = a + (b - \frac{c}{4}) \cdot n + \frac{c}{4} n^2$$

  Auch hier sehen wir, dass sich eine **quadratische Funktion** ergibt.

Schlussfolgerungen

Die sehr mühsam durchgeführte Analyse von Insertionsort war doch nicht nutzlos. Die Ergebnisse, die wir bekommen haben, sind Aussagen über das *prinzipielle Größenwachstum* der Laufzeit abhängig von der Problemgröße *n*, d.h. ob es eine *lineare* oder eine *quadratische* Funktion ist.

Auf dieser sehr groben Ebene werden wir zukünftig die Komplexität von Algorithmen analysieren. Die ganzen Details, z.B. Ausführungszeiten einzelner Anweisungen, Geschwindigkeit des Rechners etc. interessieren dabei nicht mehr. Die Ergebnisse sind dann zwar nur eine sehr grobe Beschreibung des Verhaltens, aber dafür sind es allgemein gültige Angaben, die für jeden möglichen Rechner, egal ob langsam oder schnell, gelten.

In der nächsten Lektion werden Sie lernen, wie mit Angaben über das prinzipielle Wachstum von Funktionen gearbeitet werden kann.

4.4.2 Asymptotisches Wachstum von Funktionen

Aus der Mathematik wissen Sie schon, dass eine quadratische Funktion für genügend große Werte jede lineare Funktion übersteigen wird und dass eine kubische Funktion für große Werte immer stärker wächst als eine quadratische Funktion.

Um das prinzipielle Größenwachstum von Funktionen für große Werte (für *n* gegen Unendlich, daher "asymptotisches Wachstum") vergleichen zu können, gibt es in der Mathematik die sog. *Landau-Notation* (oder auch *O-Notation*) mit den Symbolen O (Buchstabe groß-O), Ω (großes Omega) und Θ (großes Theta).

Es ist dabei zu beachten, dass nicht Funktionswerte für bestimmte einzelne Argumente verglichen werden, sondern das gesamte Verhalten von Funktionen.

Landau-Notation

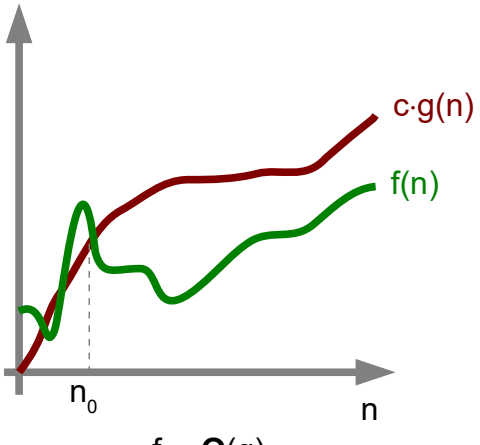
Notation	***Bedeutung***	***Beispiele***
f ∈ O(g)	f wächst nicht wesentlich schneller als g, g ist obere Schranke für das asymptotische Wachstum von f.	$30n + 7 \in O(n^2)$ $5n^2 + 9n + 13 \in O(n^2)$
f ∈ Θ(g)	f wächst asymptotisch gleich schnell wie g	$5n^2 + 9n + 13 \in \Theta(n^2)$
f ∈ Ω (g)	f wächst asymptotisch nicht wesentlich langsamer als g, g ist untere Schranke für das Wachstum von f	$2n^3 + 7n^2 \in \Omega(n^2)$ $2n^3 + 7n^2 \in \Omega(n^3)$

Formal ist die Bedeutung dieser Symbole folgendermaßen definiert:

Definition 4.23 -Asymptotische Schranken O(f), Ω(f) und Θ(f)

- **Asymptotische obere Schranke (Groß-O-Notation)**

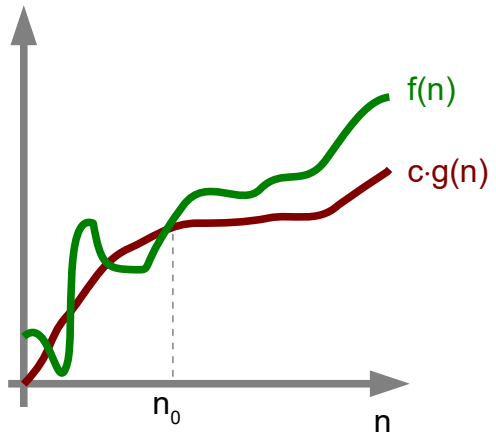
$$O(g(n)) = \{ f(n) \mid \text{es gibt ein } c > 0 \text{ und ein } n_0, \text{ so dass für alle } n > n_0 \text{ gilt } f(n) \leq c \cdot g(n) \}$$



f ∈ **O**(g)

- **Asymptotische untere Schranke (Groß-Omega-Notation)**

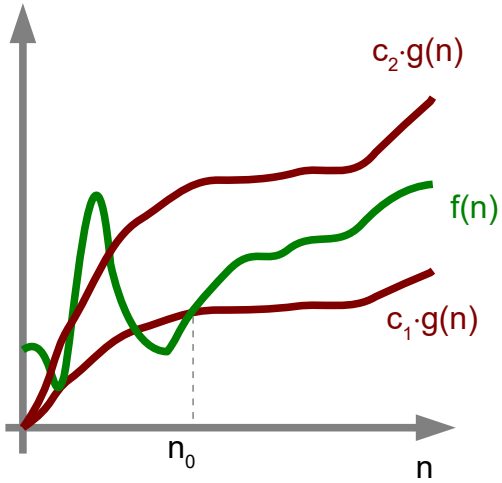
$$\Omega(g(n)) = \{ f(n) \mid \text{es gibt ein } c > 0 \text{ und ein } n_0, \text{ so dass für alle } n > n_0 \text{ gilt } f(n) \geq c \cdot g(n) \}$$



f ∈ **Ω** (g)

- **Asymptotisch äquivalent (Groß-Theta-Notation)**

$$\Theta(g(n)) = \{ f(n) \mid \text{es gibt } c_1 > 0, c_2 > 0 \text{ und ein } n_0, \text{ so dass für alle } n > n_0 \text{ gilt } c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \}$$



f ∈ **Θ**(g)

f und g seien dabei immer positive Funktionen, d.h. f(n) ≥ 0 , g(n) ≥ 0 für alle n ≥ 0.

Erläuterung

Er wird also immer nur das Verhalten für große Werte von n betrachtet, d.h. Werte, die

über irgend einer Grenze n_0 liegen, ab der das prinzipielle Verhalten dann immer gegeben ist. Bei allen drei Definitionen ist enthalten, dass konstante Vorfaktoren für das prinzipielle Verhalten, d.h. die Größenordnung, keine Rolle spielen:

- **O(*g*)**: Menge aller Funktionen die, abgesehen von einem konstanten Faktor, für große Werte von *n* nicht stärker wachsen wie Funktion *g*
- **Ω(*g*)**: Menge aller Funktionen, die, abgesehen von einem konstanten Faktor, für große Werte von *n* mindestens so stark wachsen wie Funktion *g.*
- **Θ(*g*)**: Menge aller Funktionen, die, abgesehen von einem konstanten Faktor, gleich stark wachsen wie Funktion *g*.

Anmerkung

- Man kann sich die durch **O, Ω** und **Θ** angegebenen Beziehungen als eine Art Kleiner-gleich-Relation für das Wachstum von Funktionen betrachten (hier nur als Merkregel, nicht als mathematisch exakte Definition)

  f ∈ O(g) heißt
  asymptotisches Wachstum f ≤ asymptotisches Wachstum g
  f ∈ **Θ**(g) heißt
  asymptotisches Wachstum f = asymptotisches Wachstum g
  f ∈ **Ω**(g) heißt
  asymptotisches Wachstum f ≥ asymptotisches Wachstum g

- Bei O und **Ω** ist also das gleich starke Wachstum mit dabei. Es gibt bei der Landau-Notation auch noch die Symbole o (klein-O) und ω (klein-Omega), die den Fall des gleich starken Wachstums ausschließen.

  f ∈ o(g) heißt
  asymptotisches Wachstum f < asymptotisches Wachstum g,
  "f wächst echt langsamer als g"
  f ∈ **ω**(g) heißt
  asymptotisches Wachstum f **>** asymptotisches Wachstum g,
  "f wächst echt schneller als g"

  Wir werden dies in diesem Semester aber nicht weiter verwenden.

Notationskonvention 4.24

O(g), **Ω**(g), **Θ**(g) geben nach obiger Definition jeweils eine Menge von Funktionen an. Statt f ∈ **O**(g), f ∈ **Ω**(g), f ∈ **Θ**(g) wird aber im Umfeld der Analyse von Algorithmen üblicherweise mit Gleichheitszeichen statt mit ist-Element-von-Zeichen geschrieben:

$f = \mathbf{O}(g)$
$f = \mathbf{\Omega}(g)$
$f = \mathbf{\Theta}(g)$

Aufgabe 4.25 - asymptotisches Wachstum

Stimmen folgende Aussagen?

1. $3n^2 + 500 = O(n^3)$
2. $3n^2 + 500 = \mathbf{\Omega}(n^3)$
3. $3n^2 + 500 = \mathbf{\Theta}(n^3)$
4. $3n^2 + 500 = \mathbf{\Theta}(n^2)$
5. $3n^2 + 500 = O(2^n)$

Beispiel 4.26 - Laufzeitkomplexität von Insertionsort

Die Laufzeitkomplexität für Insertionsort kann nun folgendermaßen angegeben werden (abgeleitet aus den Ergebnissen in Beispiel 4.22 - Laufzeitanalyse Insertionsort):

- **Best case**: lineare Funktion $T_{best}(n) = a + b \cdot n$

  $T_{best}(n) = \mathbf{\Theta}(n)$

- **Worst case**: quadratische Funktion $T_{worst}(n) = a + (b - \frac{c}{2}) \cdot n + \frac{c}{2} n^2$

  $T_{worst}(n) = \mathbf{\Theta}(n^2)$

- **Average case**: quadratische Funktion $T_{avg}(n) = a + (b - \frac{c}{4}) \cdot n + \frac{c}{4} n^2$

  $T_{avg}(n) = \mathbf{\Theta}(n^2)$

Insertionsort hat also im mittleren und schlechtesten Fall quadratische Laufzeitkomplexität $\mathbf{\Theta}(n^2)$, im besten Fall aber lineare Komplexität $\mathbf{\Theta}(n)$.

Obwohl Insertionsort im mittleren Fall ca. doppelt so schnell ist wie im schlechtesten Fall, ist es die gleiche Größenordnung, da sich die Laufzeiten nur um einen konstanten Faktor unterscheidet.

Für die Größenordnung ist es auch belanglos, welche Werte die Konstanten *a*, *b* und *c* in den Formeln oben haben. Um solche "Kleinigkeiten" müssen wir uns also bei der Komplexitätsanalyse zukünftig nicht mehr kümmern.

Anmerkungen

Im vorigen Beispiel wurde die Komplexität "genau" mittels Θ angegeben, d.h. "wächst gleich stark wie ...". Oft wird in der Literatur für Algorithmen die Komplexität nur mit **O**, d.h. "wächst höchstens so stark wie …" angegeben.

- Ein Grund dafür kann manchmal in der Bequemlichkeit beim Schreiben liegen, dass O statt Θ geschrieben wird.
- Neben der Bequemlichkeit gibt es aber in manchen Fällen auch "ernsthafte" Gründe, warum O(g) statt Θ(g) verwendet wird - dann, wenn die Analyse auf Abschätzungen beruht, so dass man nur sicher ist, dass es sich nicht schlechter als g verhält, es aber auch besser sein kann.

4.4.3 Häufig verwendete Größenordnungen

In der folgenden Tabelle sind die bei der Analyse von Algorithmen am häufigsten vorkommenden Komplexitätsangaben aufgeführt, angeordnet nach zunehmender Komplexität

Größenordnung	***Erläuterung***	***Beispiel***
Θ(1)	konstant	elementare Operation
Θ(log *n*)	logarithmisch	binäre Suche
Θ(*n*)	linear	lineare Suche
Θ(*n* log *n*)		schnelle Sortierverfahren
Θ(n^2)	quadratisch	einfache Sortierverfahren
Θ(n^3)	kubisch	
Θ(c^n)	exponentiell	
Θ(n!)		Permutationen berechnen

Jede der in der Liste angegebenen Größenordnungen wächst asymptotisch echt stärker als die vorangehende. (Es gilt also z.B. $n \log n = O(n^2)$, aber $n \log n \neq \Theta(n^2)$).

Logarithmisches Wachstum

Bei der Größenordnung **Θ(log *n*)** ist für den Logarithmus keine Basis angegeben. Welcher Logarithmus ist damit gemeint?

Da sich Logarithmen mit unterschiedlichen Basen nur durch einen konstanten Faktor unterscheiden (siehe Umrechnungsformel im Grundlagenkapitel), kann bei der Angabe der Größenordnungen O(log n), Θ(log n) oder Ω(log n) auf die Basis verzichtet werden (entsprechend auch bei O(n log n), Θ(n log n) oder Ω(n log n)).

Der Logarithmus zur Basis 2 wächst asymptotisch gleich stark wie der Logarithmus zur

Basis *e* oder zur Basis 10 oder zu irgend einer anderen Basis, da sie sich jeweils nur um einen konstanten Faktor unterscheiden.

$\log_b(n) = \Theta(\text{ld } n) = \Theta(\ln n) = \Theta(\lg n)$ für jede beliebige Basis b

Exponentielles Wachstum

Im Unterschied zu Logarithmen ist es bei exponentiellem Wachstum so, dass eine größere Basis zu echt stärkerem asymptotischem Wachstum führt, z.B. 3^n wächst echt stärker als 2^n, d.h.

$2^n \neq \Theta(3^n)$,

aber natürlich gilt

$2^n = \mathbf{O}(3^n)$.

Dies lässt sich folgendermaßen mittels Beweis durch Widerspruch zeigen, aufbauend auf der formalen Definition für **Θ**:

- Wir nehmen an, es würde $2^n = \Theta(3^n)$ gelten.
- Nach der formaler Definition für Θ müsste es dann eine Grenze n_0 und eine Konstante $c_2 > 0$ geben, so dass für alle $n > n_0$ gilt

  $2^n > c_2 \cdot 3^n$

  D.h. für alle $n > n_0$ muss gelten:

  $c_2 < 2^n/3^n = (2/3)^n$

Das ist aber für keine Konstante $c_2 > 0$ möglich, da $(2/3)^n$ für große Werte von *n* gegen 0 konvergiert. Die Annahme muss also falsch gewesen sein.

4.4.4 Rechnen mit Größenordnungen

Zukünftig werden wird immer mit der Landau-Notation arbeiten, wenn wir die Komplexität von Algorithmen beschreiben. Dazu sollten Sie folgende Eigenschaften kennen, um mit **O, Ω** und **Θ** richtig umgehen zu können.

Eigenschaft 4.27 - Reflexivität

- $f = \mathbf{O}(f)$
- $f = \mathbf{\Omega}(f)$
- $f = \mathbf{\Theta}(f)$

Jede Funktion ist asymptotische Obergrenze oder Untergrenze zu sich selbst und wächst asymptotisch gleich stark wie sie selbst.

Eigenschaft 4.28 - Transitivität

- Wenn f = **O**(g) und g = **O**(h) dann auch f = **O**(h)
- Wenn f = Ω(g) und g = Ω(h) dann auch f = Ω(h)
- Wenn f = Θ(g) und g = Θ(h) dann auch f = Θ(h)

Eigenschaft 4.29 - Symmetrie

- f = Θ(g) genau dann, wenn g = Θ(f)
- f = **O**(g) genau dann, wenn g = Ω(f)

Wenn g Obergrenze für f ist, ist f Untergrenze für g.

Eigenschaft 4.30 - Asymptotische Äquivalenz und obere/untere Schranke

- f = Θ(g) genau dann, wenn f = **O**(g) und f = Ω(g)

(analog wie es bei Zahlenwerten gilt: $x \leq y$ und $y \geq x$, gdw. $x = y$)

Eigenschaft 4.31 - Unabhängigkeit von konstanten Faktoren

Für beliebige Konstante c > 0 gilt:

- Wenn f = Θ(g) dann auch c·f = Θ(g) und f(n) = Θ(c·g)
- Wenn f = O(g) dann auch c·f = O(g) und f(n) = O(c·g)
- Wenn f = Ω(g) dann auch c·f = Ω(g) und f(n) = Ω(c·g)

D.h. ein konstanter Vorfaktor links oder rechts ändert nichts an der Größenbeziehung.

Eigenschaft 4.32 - Größenordnung von Summen

- falls g = O(f) dann gilt O(f + g) = O(f)
- falls g = Ω(f) dann gilt Ω(f + g) = Ω(g)
- falls g = Θ(f) dann gilt Θ(f + g) = Θ(f)

Bei Summen gilt gewissermaßen das Maximum-Prinzip: Die stärker wachsende Funktion ist entscheidend. Summiert man zwei Funktionen mit gleichem asymptotischem Wachstum, bleibt das Wachstum gleich.

Eigenschaft 4.33 - Größenordnung von Produkten

- O(f · g) = O(f) · O(g)
- Ω(f · g) = Ω(f) · Ω(g)
- Θ(f · g) = Θ(f) · Θ(g)

Beispiel 4.34

Für $2n^2 + 8 = O(n^2)$ und $4n - 1 = O(n)$, ergibt sich:

$(2n^2 + 8)(4n - 1)$

$= O(n^2) \cdot O(n) = O(n^3)$

Eigenschaft 4.35 - Polynomiale Komplexität

- Der Wachstumsgrad eines Polynoms ist durch die höchste Potenz bestimmt:

  $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_2 n^2 + a_1 n + a_0 = \Theta(n^k)$ (für $a_k > 0$)

- Höhere Potenzen wachsen asymptotisch echt stärker als niedrigere

  $n^k = O(n^j)$ für jedes $j \geq k$

  $n^k \neq \Theta(n^j)$ für jedes $j > k$

- Exponentielles Wachstum ist echt stärker als jedes polynomiale

  $n^k = O(c^n)$ für jede Potenz $k > 0$ und jede Basis $c > 1$

  $n^k \neq \Theta(c^n)$

Aufgabe 4.36 - Rechnen mit Größenordnungen

- Richtig oder falsch? Begründen Sie Ihre Antwort.

1. $3n^3 - 4n + 5 = \Theta(n^3)$
2. $100n + 99999 = \Theta(n)$
3. $3n^3 - 4n + 5 = O(n^4 + 2n^2)$
4. $3n^3 - 4n + 5 = \Omega(n^4 + 2n^2)$
5. $(3n^2 - 4n + 5)(10n^2 + 8^n + 13) = \Theta(n^4)$
6. $10 \cdot \log_2(n) + 500 \cdot n^{10} + 0{,}00001 \cdot 2^n = \Theta(2^n)$

Fazit zu Lektion 6

Das sollten Sie nach dieser Lektion beantworten können

- Was bedeutet *Komplexität* im Zusammenhang mit Algorithmen?
- Was ist eine Random-Access-Machine (RAM)?
- Welche Grundfälle werden bei der Komplexitätsanalyse üblicherweise betrachtet?
- Wie werden Größenordnungsangaben verwendet, um die Komplexität von Algorithmen zu beschreiben?
- Was ist die Bedeutung von **O**, **Ω** und **Θ**
- Welches sind die typischen Größenordnungen, die bei der Analyse von Algorithmen vorkommen?
- Wie kann mit den Größenordnungen **O**, **Ω** und **Θ** gerechnet werden?